Optimizing Sequential Key Inserts into B-Tree

Abstract

B-Trees are one of the most widely used data structures in database systems, used to index data from a variety of sources. As servers running RDBMS's have increased the number of processors available, the demand for data structures that can support concurrent operations has increased along with it. Many extensions of B-Trees have been proposed that minimize locking (**FIXME** e.g. Latch Coupling, B^{link}-Link Trees) or eliminate locking completely (**FIXME** e.g. Bw-Trees) to improve concurrency. While these methods have greatly increased to concurrent throughput over baseline locking algorithms, concurrent sequential key inserts are still problematic for many algorithms, and lead to greatly reduced throughput compared to other workloads. In this paper we examine the problem of sequential key inserts and propose a solution that extends a B-Tree Optimistic Lock Coupling to reduce contention between threads and improve throughput of this typically challenging workload.

Background

Related Work

Bw-Trees

B-Tree with OLC

Other Tree Structures

Problem

Consider the following workflow, 1. A customer submits an order 2. The server adds additional information about the order and gets the next unique order id (monotonically increasing) 3. The server then insert the completed order into the database 4. The server then notifies the customer that the order is placed

In this scenario it is likely that the order ids will be indexed as they make an ideal primary key. For a hash index, such a workflow is not an issue since the hashing should randomly distribute the keys and hence contention between concurrent modifications should be low. For a B-Tree however this is very problematic. Because the order ids are monotonically increasing, all concurrent threads are going to be inserting into the right most leaf of the tree. For algorithms that take locks to perform updates, this is going to hurt performance since all threads need to take the same lock to insert the order. It is of course possible to randomly select order ids, turning the sequential inserts into a random inserts, however you loose the ability to quickly iterate over the orders in chronological order. For similar reasons a hash index might not be appropriate. Iterating over orders in chronological order maybe very useful when determining which orders to fill first in the event of limited supply hence maintaining a monotonically increasing order id in a B-Tree index adds functionality that is not present in other solutions.

Solution

As illustrated above, scenarios certain workloads are very problematic for current B-Tree implementations. The key observation is that most algorithms breakdown when there are many concurrent writers on the same leaf of the tree. Our solution is simple, make concurrent inserts into a single leaf as fast as possible. To do this we propose a two step solution when handling inserts into the B-Tree.

Experiments

Results

Discusssion

Sources